Recursive Time Complexity

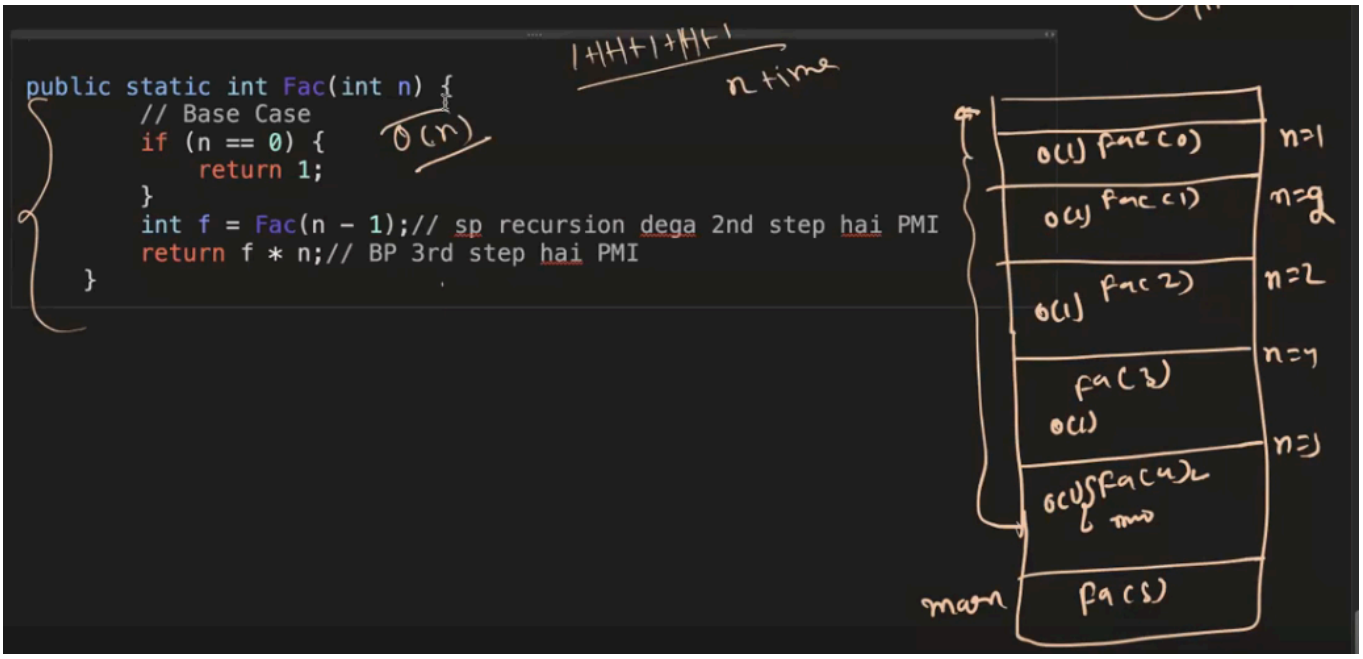
We will analyze the time complexity of recursion, focusing on two main cases:

1. When there is a single recursive call.
2. When there are multiple recursive calls.

When there is a single recursive call

Basically, to determine the time complexity of a recursive function, I look at how many recursive frames (or stack frames) are created — in other words, how many times the function calls itself. Then, I calculate the time complexity of each frame and sum them up. The total amount of work done across all recursive calls gives me the overall time complexity.

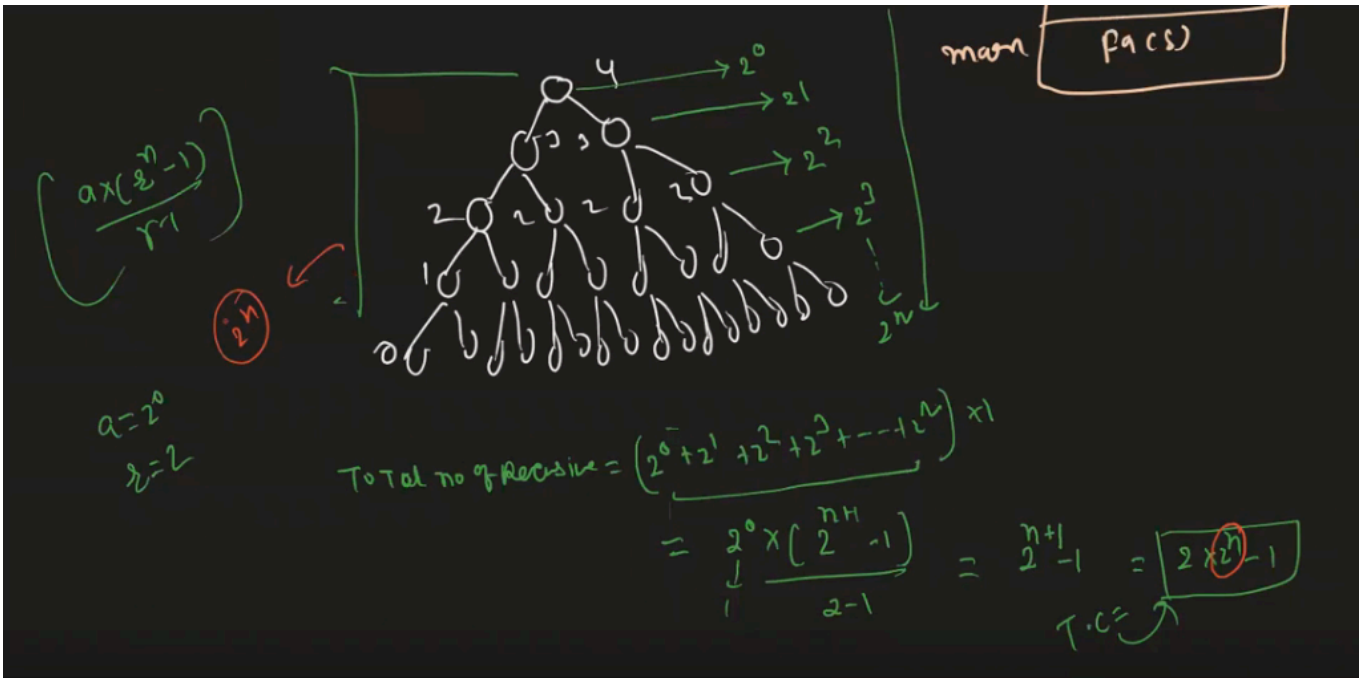
https://github.dev/marvel2950/Java-DSA-1.0/tree/main/Day%2015



When there are multiple recursive calls

Number of recursive calls to the power n.

https://github.dev/marvel2950/Java-DSA-1.0/tree/main/Day%2016



🎯 Power logN

```java
public class Main {
	public static void main(String[] args) {
		int a = 3;
		int n = 4;
		System.out.println(pow(a, n));
	}

	public static int pow(int a, int n) {
		if (n == 0) {
			return 1;
		}
		int ans = pow(a, n / 2);
		ans = ans * ans;
		if (n % 2 != 0) {
			ans = ans * a;
		}
		return ans;
	}

}
```

📘 What is a Recurrence Relation?

Function of time which helps in finding time complexity

🧮 Recurrence Relation for Factorial

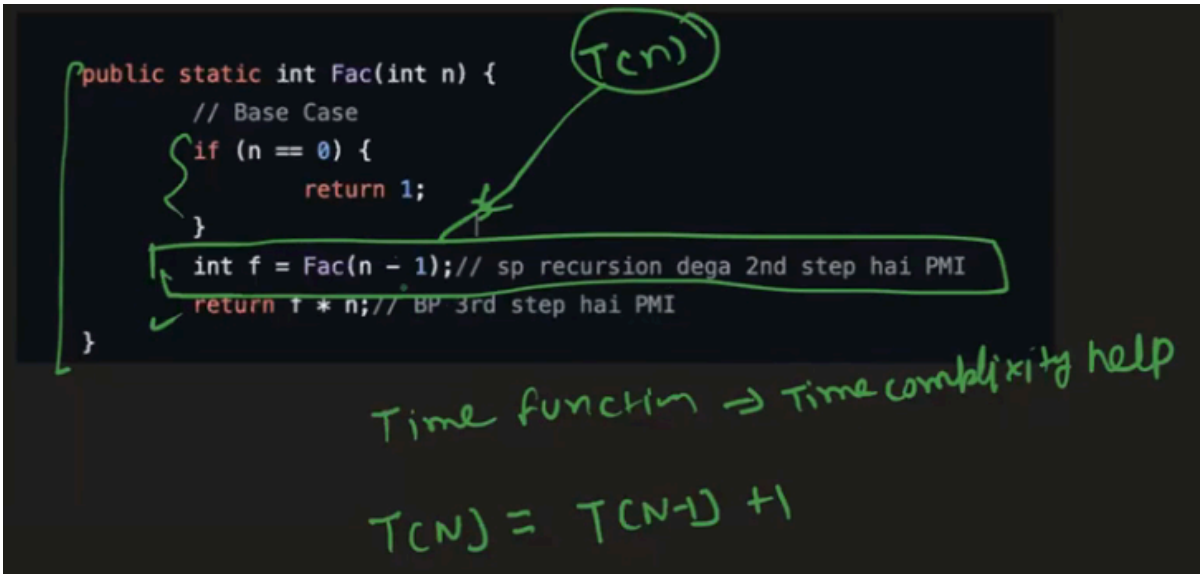
When analyzing the time complexity of computing the **factorial of n**, we consider how the function behaves recursively.

To compute `factorial(n)`, we must first compute `factorial(n - 1)`. This means the time taken to compute `factorial(n)` depends on the time taken to compute `factorial(n - 1)`, plus a constant amount of work (typically the multiplication and function call).

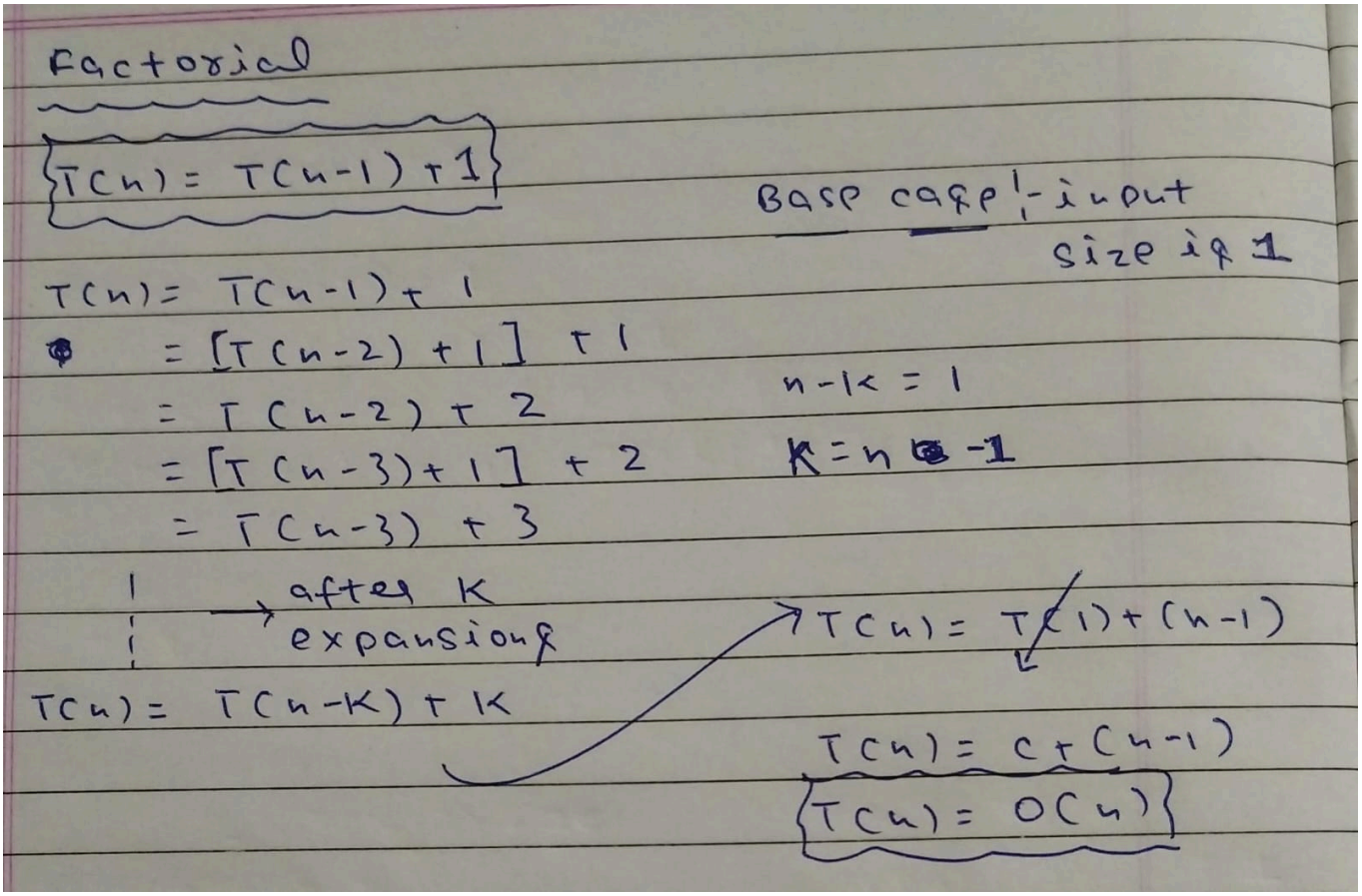
So, the **recurrence relation** becomes:

T(n) = T(n - 1) + 1

This indicates that each recursive step takes constant time (`+1`), and the problem size reduces by 1 with each step, until the base case is reached (usually `T(1)` or `T(0)`).



🔁 Expansion Method (for Solving Recurrence Relations)



Binary Search

$$T(n) = T\left(\frac{n}{2}\right) + 1$$

Base case :-
input size is 1.

$T(n) = T(n/2) + 1$
$= T(n/4) + 1 + 1$
$= T(n/8) + 1 + 1 + 1$
⋮ → after k expansions
$T(n) = T(n/2^k) + k$

$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

$T(n) = T(1) + \log_2 n$
$T(n) = C + \log n$
$T(n) = O(\log_2 n)$

merge Sort

$$T(n) = 2 \cdot T\left(\frac{n}{2}\right) + n$$

Base case :-
input size is 1

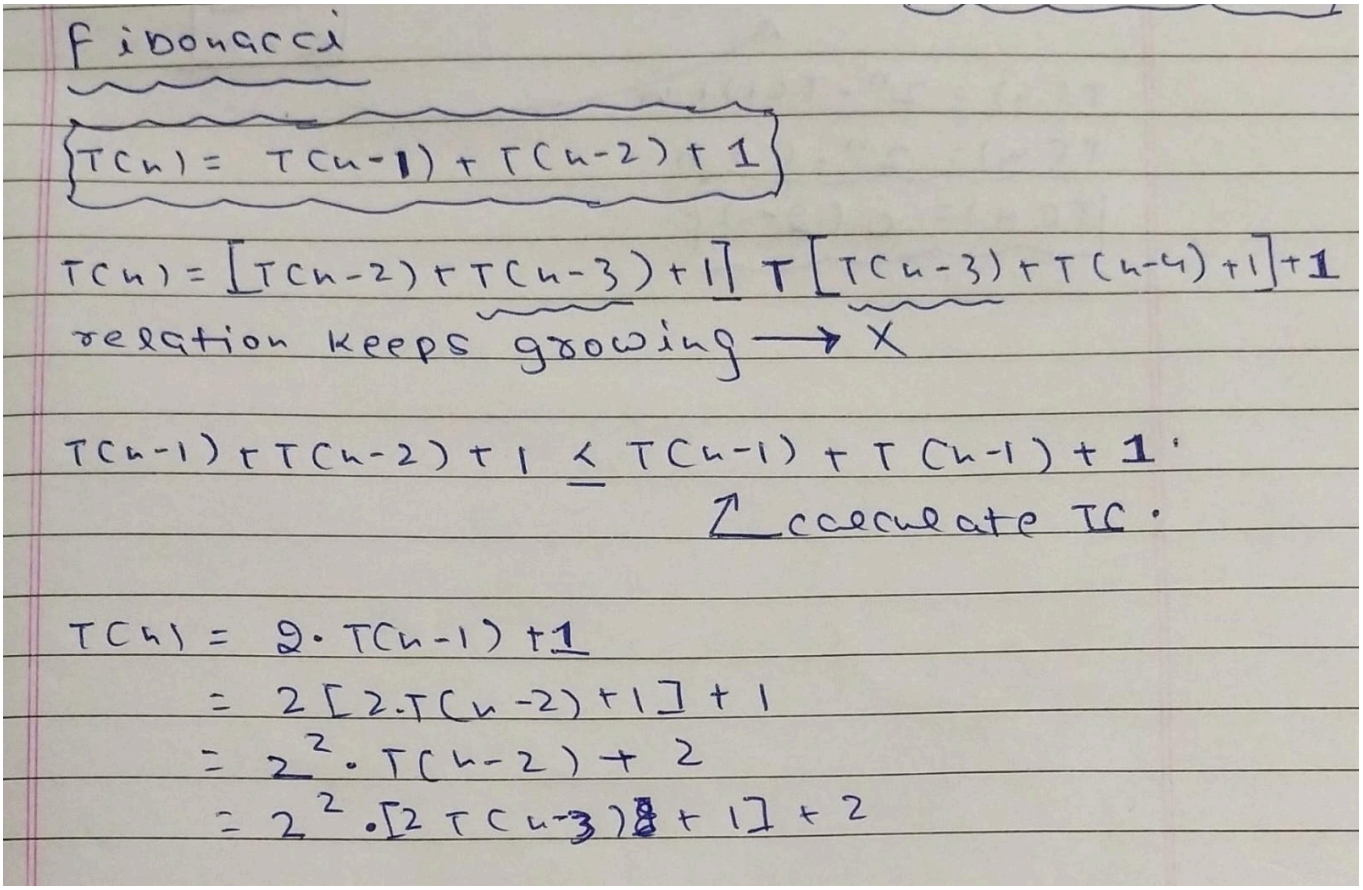
$T(n) = 2T(n/2) + n$
$= 2[2T(n/4) + n/2] + n$
$= 4T(n/4) + n + n$
$= 4[2T(n/8) + n/4] + 2n$
$= 8T(n/8) + 3n$.
⋮ → after k expansions
$T(n) = 2^k T(n/2^k) + k \cdot n$

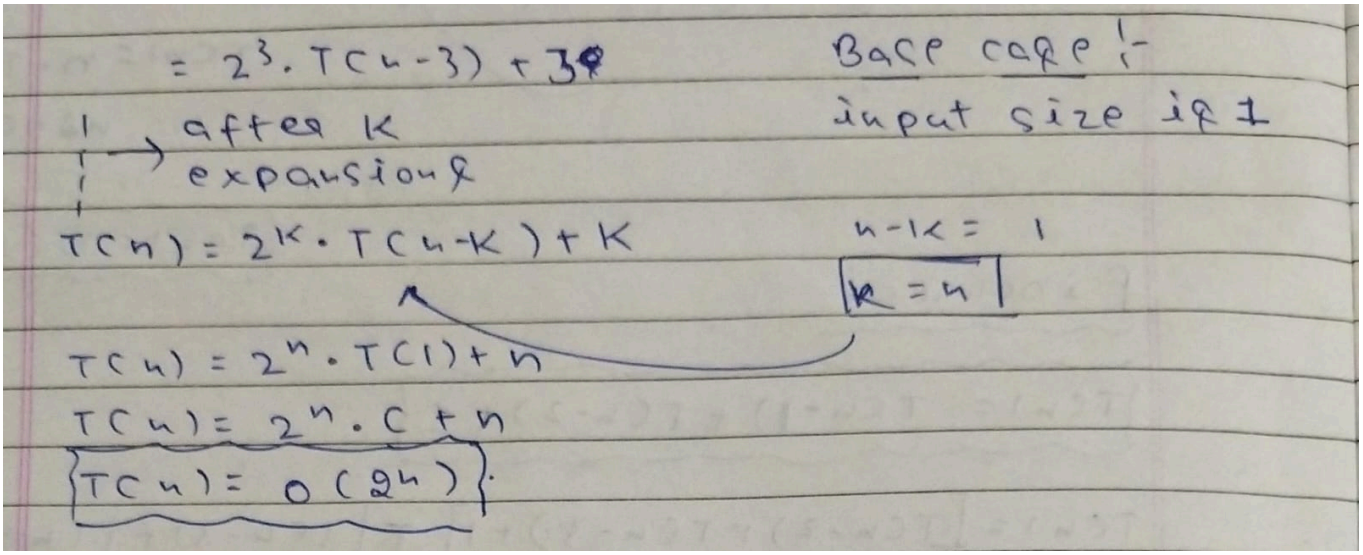
$\frac{n}{2^k} = 1 \Rightarrow k = \log_2 n$

$T(n) = 2^{\log n} \cdot T(1) + \log n \cdot n$

$T(n) = n \cdot T(1) + n \log n$

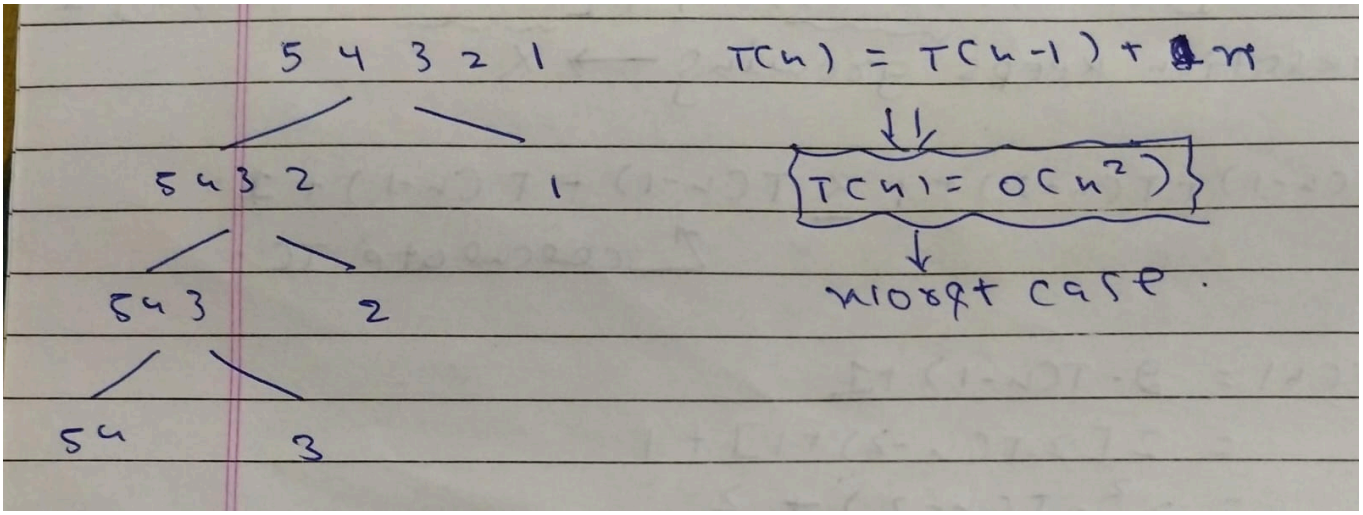
$T(n) = O(n \log n)$





🎲 What is Randomized Quick Sort?

Randomized Quick Sort is a variation of the classic Quick Sort algorithm where the **pivot element is chosen randomly** instead of using a fixed position (like the first or last element).



Why Randomize the Pivot?

- **Classic Quick Sort** can perform poorly (worst-case O(n^2)) if the pivot is consistently the smallest or largest element, especially on already sorted or nearly sorted data.
- **Randomizing the pivot** reduces the chance of consistently poor splits.
- This leads to **better average-case performance** and makes the algorithm's runtime more predictable and closer to the average O(nlogn).

How Randomized Quick Sort Works:

1. **Choose a pivot randomly** from the current array/subarray.
2. **Partition** the array into two parts:
    - Elements **less than** the pivot.
    - Elements **greater than or equal to** the pivot.
3. Recursively apply Randomized Quick Sort to both parts.
4. Combine the results.

Time Complexity:

- **Average case:** O(nlogn)
- **Worst case:** O(n^2), but with very low probability due to random pivot selection.

```
public class Main {
	public static void main(String[] args) {
		int[] arr = { 5, 7, 2, 1, 8, 3, 4 };
		sort(arr, 0, arr.length - 1);
		for (int i = 0; i < arr.length; i++) {
		   System.out.print(arr[i] + " ");
		}
	}

	public static void sort(int[] arr, int si, int ei) {
		if (si >= ei) {
			return;
		}
		int idx = partition(arr, si, ei);
		sort(arr, si, idx - 1);
		sort(arr, idx + 1, ei);

	}

	public static int partition(int[] arr, int si, int ei) {
```

```
            Random rn = new Random();
            int ri = rn.nextInt(ei - si) + si;

            int tt = arr[ei];
            arr[ei] = arr[ri];
            arr[ri] = tt;

            int item = arr[ei];
            int idx = si;
            for (int i = si; i < ei; i++) {
                  if (arr[i] <= item) {
                        int temp = arr[i];
                        arr[i] = arr[idx];
                        arr[idx] = temp;
                        idx++;
                  }

            }
            int temp = arr[ei];
            arr[ei] = arr[idx];
            arr[idx] = temp;
            return idx;
      }

}
```